P2P-RPC: Programming Scientific Applications on Peer-to-Peer Systems with Remote Procedure Call

Samir Djilali
Laboratoire de Recherche en Informatique
UMR 8623 CNRS - Paris-Sud University
91405 ORSAY Cedex - France
djilali@lri.fr

Abstract

This paper presents design and implementation of a remote Procedure call (RPC) API for programming applications on Peer-to-Peer environments. The P2P-RPC API is designed to address one of neglected aspect of Peer-to-Peer - the lack of a simple programming interface. In this paper we examine one concrete implementation of the P2P-RPC API derived from OmniRPC (an existing RPC API for the Grid based on Ninf system). This new API is implemented on top of low-level functionalities of the XtremWeb Peer-to-Peer Computing System. The minimal API defined in this paper provides a basic mechanism to make migrate a wide variety of applications using RPC mechanism to the Peer-to-Peer systems. We evaluate P2P-RPC for a numerical application (NAS EP Benchmark) and demonstrate its performance and fault tolerance properties.

1 Introduction

The main goal of Peer-to-Peer programming is the study of programming models, tools and methods that support the effective development of high-performance algorithms and applications on Peer-to-Peer environments. Programming applications on top of Peer-to-Peer systems will required properties beyond that of simple sequential programming or even parallel and distributed programming. Besides managing simple operations over private or distributed data structures, programmer of applications to be run on a Peer-to-Peer system will have to deal with an environment that is typically open-ended, heterogeneous and dynamic. The programming model must give those heterogeneous and dynamic resources a common "look-and-feel" to the programmer. This transparency that should be provided by the run-time system is a paramount condition to facilitate programming for Peer-to-Peer systems. However, it would be necessary to keep in mind that Peer-to-Peer programming is restricted to a limited applications scope. A message passing API (MPICH-V) [1] has been proposed for the Peer-to-Peer environments, but the weak communication performances make it difficult to consider applications with high communication/computation ratio. On the contrary, parameter sweep, bag of tasks and Master-Worker applications are suitable to such environments.

2 Related Work

The concept of Remote Procedure Call (RPC) [2] has been used for a long time in distributed computing as it provides a simple way to allow communication between distributed components. Most of the previous works have focused on the development of high performance RPC mechanisms and RPC for the Grid.

In Peer-to-Peer computing environments, high performance can be expected in application and run-time environment fulfill several constraints: Fault tolerance, Adaptation to the peer group size, and Adaptation to the peer's available resources capacities (memory, disc, communication,...).

Currently, RPC is used as an enabling programming paradigm for building Peer-to-Peer platforms (Web Services[3], SOAP[4], JXTA[5], ...) but not as a programming paradigm for applications built on top of the infrastructure.

To our knowledge, no programming model exists for Peer-to-Peer systems. This is due to the fact that, actually, Peer-to-Peer systems are mostly used for file sharing [6, 7] and not for deploying scientific applications. Scientific applications that consume large computing power are generally intended to dedicated machines or Grids, because these environments are better controlled. This is why several programming environments has been proposed for the Grid. The most known are GridRPC[8] and OmniRPC[9].

GridRPC is a proposal to standardize a remote procedure

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

call (RPC) mechanism for Grid computing. It implements an API (Application Programming Interface) on two different Grid computing systems: NetSolve[10] and Ninf[11]. NetSolve is a client-server system which provides remote access to hardware and software resources through a variety of client interfaces, such as C, Fortran, and Matlab. A NetSolve system consists of three entities: a) the *client*, which needs to execute some functions remotely, b) the *server* executes functions on behalf of the clients and c) the *agent* maintains a list of all available servers and performs resource selection for a client request. Ninf is another client-server based API for Grid. Its last implementation: Ninf-G uses the Globus[12] toolkit to manage the execution of a client request (remote call) on a server. Netsolve and Ninf have not been designed to handle the volatility of nodes in Peer-to-Peer systems.

OmniRPC is another RPC based programming environment for cluster and Grid computing. OmniRPC automatically allocates calls dynamically on appropriate remote computers. It also support parallel programming (multi-threaded) by allowing client to issue multiple requests on different remote computers simultaneously.

We can also cite the CondorMW [13] based on master-worker paradigm. The master distributes computation to condor connected workers when they are idle. The master has to manage worker loss by assigning their jobs to others available workers. The most significant inconvenient of this system is that it is not robust in the presence of failure of the master.

Such environments propose a programming model based on RPC that not feet Peer-to-Peer paradigm. As P2P-RPC is intended to be used for parallel applications and to provide a fault tolerance system for the client, an asynchronous RPC layer provides a suitable programming model for such environments. Fault tolerance can be managed by the programmer from returned value of RPC calls or automatically by the P2P-RPC framework. In this paper we consider the second approach.

3 Principles of Peer-to-Peer Programming

There are several general properties that are desirable for all programming models. Properties for Grid programming models have also been discussed in [14]. The Peer-to-Peer environment presents many major challenges.

- **Portability and Adaptability:** Some current high-level languages (Java[15],.Net[16]) allow codes to be processor independent. Peer-to-Peer programming models should enable codes to have similar portability. This is a necessary prerequisite for coping with dynamic, heterogeneous configurations. Also, a Peer-to-Peer program should be able to adapt itself to different configurations based on available resources. It will be preferable to have such adaptability as transparent property of the run-time environment.
- **Network Performance:** Due to the specific nature of the network infrastructure used to deploy Peer-to-Peer environments (nodes interconnected over Internet), low bandwidth and high latency limit the performance of highly communicating applications.
  The ratio communication/computation is a key for the tasks placement in such systems. It is necessary to adapt the placement of the tasks, depending on the network performance and their communication requirements.
- **Fault Tolerance:** The dynamic nature (volatility of resources) of Peer-to-Peer systems means that fault tolerance is a significant aspect to be taken into account. For example, highly distributed codes like Monte Carlo or parameter sweep applications, should initiate thousands of simulations which are independent jobs on thousands of hosts. In this context, the system or the programmer has to manage jobs lost, by re-allocating them to other host.
- **Security:** A Peer-to-Peer environment gather several thousands of resources. It is clear that traditional login identification mechanisms are impossible in such context. But, minimum security mechanisms are required to guarantee confidentiality of data and protection of participant resources.
- **Simplicity:** To ensure the survival of a programming model, it must be simple and easy to use. The fact of being able to adapt existing applications easily, constitutes a major asset.

4 Programming Peer-to-Peer Systems with RPC

One definition of Peer-to-Peer computing is the sharing of computer resources and services by exchange between systems.

4.1 XtremWeb

We have implemented our RPCs programming interface for XtremWeb[17]. XtremWeb is an experimental Global Computing platform. The key idea of Global Computing is to harvest idle time of Internet connected computers which may be widely distributed across the world, to run a very large and distributed application. All the computing power is provided by volunteers computers, which offer some of their idle time to execute a piece of the application. Thus Global Computing extends the cycle stealing model across

IEEE COMPUTER SOCIETY

the Internet. In such environment, we can distinguish between three entities: a) the *worker* is the volunteer machine executing a task, b) the *client* is the end user requesting for some services provided by c) the *coordinator* which ensure the dialog between clients and workers, and the system management.

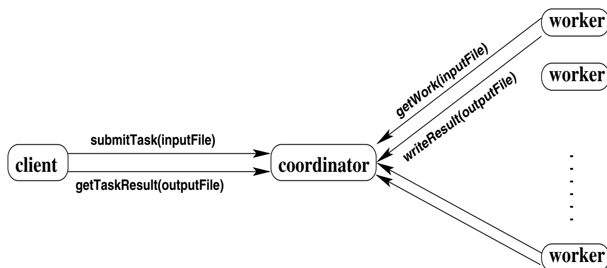

Figure 1. General organization of XtremWeb P2P environment

When the user wants to run an application on the XtremWeb platform, he has to express his application as a set of independent tasks. After building a task, the client submits it to the coordinator. On the other side, if a worker machine is idle, it contacts the coordinator to ask for job. If the coordinator has some remaining tasks in his queue, it sends it as a response to the worker machine request. The worker executes the task locally and returns the result file to the coordinator when it finish. As soon as results are available on the coordinator side, they can be sent to the user. This communication mode (all communications are initiated by client or worker) allows an easier deployment bypassing fire-walls blocking incoming request from the server located outside of the administrative domain. This protocol is independent of the communication layer.

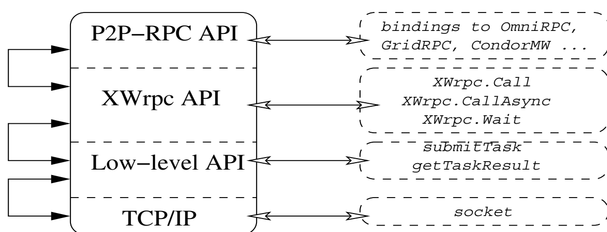

Figure 2. Different client API layers

Figure 2 shows the different levels of interfaces available in XtremWeb system. The low-level API is a set of basic functions allowing the communication between client and coordinator. Its main goal is to permit a client submit an XtremWeb task and retrieve its result from the coordinator. XWrpc API is a set of functions implementing a RPC mechanism by using XtremWeb low-level functions. This new interface is more efficient to use for programming applications where it provides a limited set of simple to use functions.

The two following subsections describe the implementation of these two interfaces (low-level and XWrpc API) and the manner to use them for programming applications on XtremWeb.

4.2 XtremWeb Low-level API

Here is the outline of an example using XtremWeb's low-level API. It is a master/worker implementation of EP (Embarrassingly Parallel) benchmark from the NAS NPB-2.3 suite. This new version is widely inspired from the MPI one. The master initiates the EP computation, spawn several tasks and submit them to the XtremWeb coordinator. When all tasks are done, the master retrieves all results from the coordinator and make the final reduction.

```
main(String [] args) {
 connect(clientId, coordinator);
 sid = createSession();
 gid = createGroup(sid);
 for(k=0;k<nbNodes;k++) {
   /* build task k */
   tid[k]=submitTaskInGroup(gid,task[k]);
 }
 k = 0;
 while (k < nbNodes)
   k = k + getGroupResult(gid);
 /* processing result files */
 /*and making final reduction */
 ...
 deleteSession(sid);
}
```

First, the client have to open a connection with the coordinator. If it is successfully identified by the coordinator it can open a work *session*. In a session, the client have two possibilities: 1) create *groups* and submit his tasks in a group, or 2) submit his tasks in the session. In the first case, the tasks can be retrieved even the connection with coordinator was closed. At contrary, all tasks submitted in the session will be lost if the client is disconnected before retrieving results i.e. we have two modes of communication between the client and the coordinator: *connected* and *not connected*.

A program written in XtremWeb low-level API comports three main steps: initializing run-time environment, submitting tasks and retrieving results, and finalizing. In the following, we informally describe the XtremWeb programming model and the functions that comprise the low-level API. Most of these functions are used in the program above.

0--7695-1919-9/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

- **Initializing and Finalizing Functions:**
  - connect(clientId, coordinator) and disconnect(): establish or release the communication link between client and coordinator.
  - createSession(), createGroup(sessionId), deleteSession(sessionId) and deleteGroup(groupId): managing sessions and groups of tasks. note that deleting a session causes the loss of all groups and tasks created during it. Deleting a group causes the loss of all its tasks.
- **Sending Tasks and Retrieving Results:**
  - submitTaskInGroup(task) and submitTaskInSession(task): send a task, to be executed by a volunteer worker, to the coordinator. According to the function, the task will be included in a session or a group. These functions return immediately after submitting tasks and do not wait for task results.
  - taskStatus(taskId), groupStatus(groupId) and SessionStatus(sessionId): return execution state of a task, a group or a session.
  - getTaskResult(taskId), getGroupResult(groupId) and getSessionResult(sessionId): retrieve results of submitted tasks. These functions blocks until, at least, one result is available.
- **Configuring Execution Environment:**
  - addApplication(app) and removeApplication(appId): allow to add or remove a client codes (binaries or byte code) to the set of applications registered at the coordinator.
  - setWorkersNb(int): defines the number of computers to use during a session.

4.3 First Implementation of P2P-RPC

In this section, we propose an API for a RPC mechanism dedicated to Peer-to-Peer systems. The first implementation has been done on top of XtremWeb platform as its low-level API is well adapted to RPC programming model.

Our API extends OmniRPC's one by changing radically its runtime environment. OmniRPC is dedicated to Grid problematic: a client connects to a resource manager which send association lists between network hosts and services. Then, the client choose locally the host which compute the RPC result.

In our runtime environment, the client request a RPC execution. A coordinator manages workers in order to complete the request. The client is not aware of the location of the computation host and get back the result in a finite time.

This kind of behavior provides three new functionalities: Fault-tolerance, Asynchrony, and Connection-less communications.

To illustrate the use of our P2P-RPC model, we have ported the parallel version of EP (subsection4.2) on XtremWeb Peer-to-Peer system, using XWrpc interface. The master initiates the run-time environment, launch workers and get results using RPC calls, make the final reduction, then finalize the computation. This program uses asynchronous call in order to benefit from parallelism: many tasks are launched in parallel on different workers according to available resources in the system.

```
main(String [] args){
 XWrpc.Init(args);
 for(k=0;k<nbNodes;k++)
  XWrpc.CallAsync(``ep'',k,nbNodes,results[k]);
 XWrpc.WaitCallAsyncAll();
 /* making EP final reduction */
 ...
 XWrpc.Finalize();
}
```

Most of functions used in this program are described hereafter. We detailed how XWrpc interface is implemented on top of XtremWeb low-level API.

4.3.1 XtremWeb RPC API

The API is divide in two parts:

- A common part with OmniRPC including the three standard functions of the RPC mechanism:
  - **XWrpc.Init(initializing arguments)**: Initialize the run-time environment. This consist to call these low-level functions: connect(clientId, coordinator); sessionId = createSession(); groupId = createGroup(sessionId); in order to connect the coordinator and create a default session and group.
  - **XWrpc.Call(function, input and output parameters)**: Call the specified function with the given parameters, and blocks until the call results are coming back to the client side. Remote functions are implemented as executable files. The remote call is simply implemented by using the low-level API described above: first, a task is build by describing the function to execute and it's parameters (the

IEEE COMPUTER SOCIETY

function is described by the executable name, and the parameters of the call are gathered in an input file for the task) ; then the task is submitted to the coordinator by using tid = submitTaskInSession(sessionId, task). Submit a task in a session means that the call result must be retrieved before the end of the session. As XWrpc.Call is a synchronous call, then we ask for the task result by using getTaskResult(tid) which blocks until the file result is coming on. After what, a handler is called automatically to process the result file and fill in the output parameters.

– `XWrpc.Finalize()`: Free all allocated resources. This can be done by cleaning the coordinator database and closing the connection with it: deleteSession(sessionId); disconnect(); are the low-level functions used in this case.

These functions correspond to the *OmniRPC_init, OmniRPC_call* and *OmniRPC_finalize* functions.

- An asynchronous part:
  – `XWrpc.CallAsync(function,input and output parameters)`: Call the specified function with the parameters. This function send the request to the coordinator and returns immediately, notifying the acceptance and giving back an identifier for the call. This function is implemented like the synchronous one but with two deferences: the task is submitted in a group (tid = submitTaskInGroup(groupId, task);) in order to benefit from the non-connected mode ; and it will not ask for the result but return immediately after submitting the task.
  – `XMrpc.WaitCallAsync(call identifier)`: This function blocks until the result of the call specified by the identifier is available. Then, the function return the result via the output parameters. This is implemented by calling the low-level function getTaskResult(tid) which will block until the result file is back. Then a handler is called to process result file and deliver output parameters of the asynchronous RPC call.
  – `XWrpc.WaitCallAsyncAll()`: This function is equivalent to execute a XWrpc.WaitCallAsync() on every pending asynchronous calls. This is done by calling the low-level function getGroupResult(groupId) several times until all results are available at the client side. Every time getGroupResult returns some results files, a handler is automatically called in order to proceed them and provide the asynchronous call output parameters.

4.3.2 Properties of P2P-RPC

In our implementation of RPCs, RPC request is posted as a task of XtremWeb, and RPC answer is to be taken as a result of XtremWeb. This means that an RPC is divided into two phases: 1) asking for a service by sending an XtremWeb task to the coordinator and 2) getting the results of the remote service by retrieving file result from the XtremWeb coordinator. This schema implies many properties of our P2P-RPC:

- **Connection-less:** Because sending a call and receiving results are de-coupled, the client can disappear or close connection with coordinator without any loss of RPC results. The results will stay stored in the coordinator side, and the client can request results of its pending calls at the next re-connection, without resubmitting any previous call.

- **Asynchrony:** Client can submit many calls before asking for results. Such calls consist to send an XtremWeb tasks to the coordinator (first phase of a RPC). In this mode, calls can be executed in parallel, each one on a worker, according to resources availability.

- **Recursivity:** Our RPC implementation allows user to easily program recursive applications, which is a highly used paradigm for a class of large scale optimization algorithms (Branch and X). This means that a remote procedure can itself launch a RPC requests by using the same API. Here is an example illustrating this operating mode:

```
function Factorial(in n, out result)
{
  if ( n == 1) result = 1;
  else {
   XWrpc.Call(Factorial, (n-1), r);
   result = n * r;
  }
}
```

  When the client launch its call (`XWrpc.Call(Factorial, m, r)` for example), a task will be send to the coordinator. An available worker (lets call it w1) will ask the coordinator for a job and get the initial task (Factorial(m,r)). When running the task, w1 will launch another client c2 to compute `Factorial(m-1,r1)`: a task is send to the coordinator. Another worker w2 will get this task and

IEEE COMPUTER SOCIETY

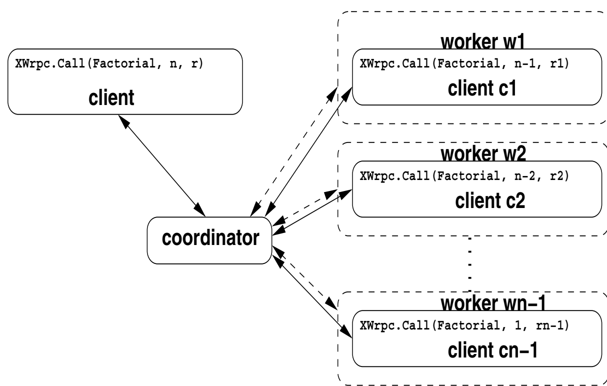

Figure 3. A recursive execution using XWrpc

compute the remote call by launching another client if necessary and so on. At the end of recursivity process, when c2 receives its call result, w1 sends a result file to the coordinator. Then the initial client can get its results.

- **Fault-Tolerance:** Fault tolerance is an important issue in Peer-to-Peer environment because resources are very volatile. In our implementation of P2P-RPC, fault-tolerance is insured by the underlaying environment (XtremWeb) and is completely transparent to the client i.e when the client sends its RPC request he is sure that he will get the results of the call even if some failures happened, except if all the system is down. Indeed, any entity (client, coordinator or worker) of the Peer-to-Peer system can disappear without affecting the execution (but the performance) of a RPC for the client point of view: **a)** The coordinator stores information concerning tasks and workers on reliable media (disk). On start (or restart), the coordinator reads information stored on disk to setup its proper state. **b)** If the worker is lost (detected by a time-out mechanism) then its work is scheduled to another available worker, by the coordinator. If the worker was running a client, then its client is also rescheduled to another worker. **c)** If a client disappear then it can be re-launched (on the same or another machine) and can retrieve results of all RPCs launched previously without re-launching them (all results stay stored at the coordinator side).

4.4 Examples

4.4.1 NAS EP Benchmark

Here, we present the results of running EP Benchmark - class C on XtremWeb platform at LRI using a coordinator on a bi-processor PIII 700MHz (256 MB RAM) and workers and a client on AMD 1.5GHz (500MB RAM) PCs. All these PCs are connected over switched Ethernet 100 Mb/s network. We decomposed EP in 100 tasks. EP Benchmark was implemented with XWrpc using strategy described in subsection 4.3.

We have also measured the overhead of the system when

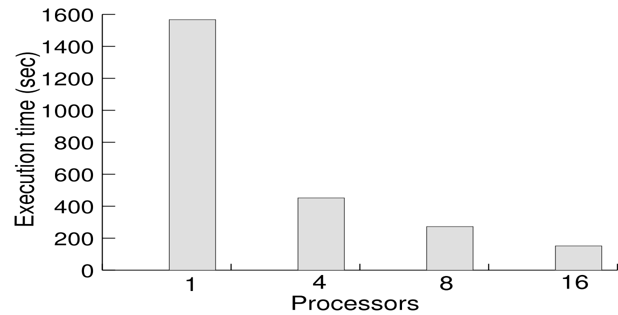

Figure 4. Execution time of EP benchmark using XWrpc

submitting one RPC for EP Benchmark. The input parameters where two integers and the output ones where 13 float numbers. The local computation time was 15 seconds and the overhead was 3 seconds. This overhead includes the time to transfer the input file to the coordinator, save this file in the coordinator side, send this file to the worker, zipping the results file on the worker, sending it to the coordinator, retrieving this file from the coordinator to the client, and unzipping it on the client side to get output parameters of the initial remote call. The overhead is quite significant if we consider execution duration of seconds on the worker side. However, if we consider execution of suites of tasks, this overhead is negligible.

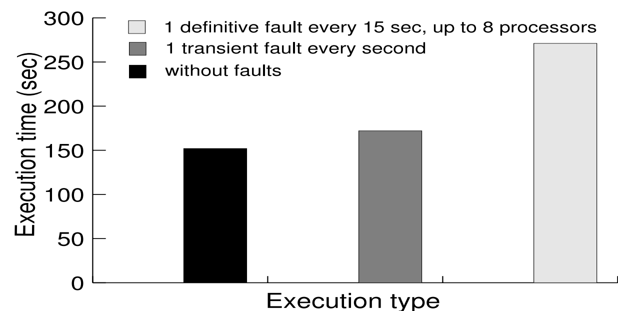

Figure 5. Execution time of EP benchmark in faulty environment

Figure 5 shows the execution times of EP benchmark - class C on 16 processors (workers). We have study two kind of faults: transient and definitive. For the first one, every second, a worker disappear for few seconds (2 - 3 seconds) then it re-connect the system. This causes the loss of a task

IEEE COMPUTER SOCIETY

every second. The second kind of faults is definitive, when a worker crashes, it will never re-contact the coordinator during the experience (this implies that after detecting this fault, the coordinator has to re-schedule the task held by the died worker to another available one). A crash happened every 15 seconds up to 8 worker (50% of workers loss). Transient and definitive faults causes respectively 12% and 78% of time overhead. We remark that the degradation of performance is not very important in transient faults case. This is mainly due to the fact that a worker saves its last running job (on disk), and re-execute it when it re-connect the system ; this avoids useless transfers between worker and coordinator (in order to get a new job to run, for example).
If we take in account that these faults are handled automatically by the system and no programming efforts are needed from the programmer, then this overhead stay reasonable.

4.4.2 Recursive function: Factorial

On the same platform, at the LRI laboratory, we have tested a recursive programm (Factorial) written with XWrpc. In this example, every task *Factorial(n)* generates a new one *Factorial(n-1)* until the last one *Factorial(1)*. In this scheme, every call is blocked until receiving its results (the results of the child task). This is an adequate example to proof the feasability of recursive programming model on Peer-to-Peer systems.

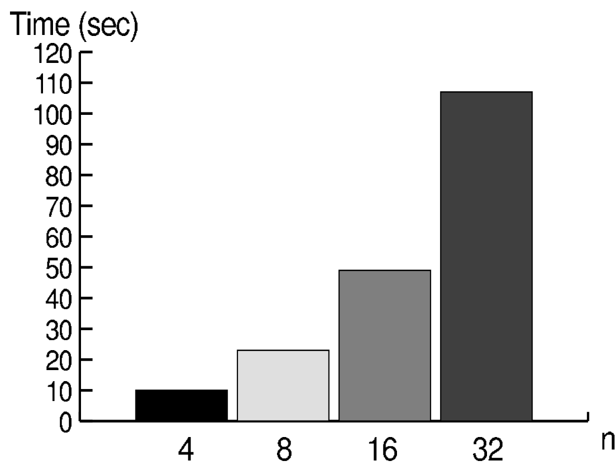

Figure 6. Execution time of recursive Factorial(n) function using XWrpc

Figure 6 shows execution times of Factorial example. These times represent primarily the underlaying system overhead. Because every call issued another and do not make any computation. It confirms the overhead calculated before ($\tilde{3}$ seconds per call).

5 Conclusion

We have presented the design of an API for a Peer-to-Peer programming inspired from similar works for the Grids. Considering the collaboration work scheme in Peer-to-Peer environment and more especially in XtremWeb, it is completely natural to use Remote Procedure Calls in such conditions. Indeed, a client send a request as a couple (binary name, input data file) to the coordinator. This task will be scheduled to a worker who will return a result (output data file) to the coordinator. Then, the client can retrieve his result (data output file) from the coordinator. In a RPC mode, a client request for a function by calling it with the input parameters. and get results in the output parameters. function name, input parameters and output parameters, of a RPC, corresponds respectively to binary name, data input file and data output file in Peer-to-Peer system.

As a future work, we plan to implement binding from our P2P-RPC API to other known API used in Grid environment like GridRPC or Condor Master-Worker. These programming models (RPC and Master/Worker) are very promising for Peer-to-Peer computing environments, because of their limited needs in communications. Exploring combinative optimization applications with using recursivity is another interesting kind of applications that can use our RPC mechanism.

Another important issue, is fault tolerance. We project to make our RPC mechanism fully fault tolerant by using coordinator replication, checkpoints and message logging techniques.

6 Acknowledgments

The author thanks Mitsuhisa Sato for issuing the challenge that led to this paper, Franck Cappello for his helpfull suggestions for improving this paper, George Bosilca and Frédéric Magniette for reviewing drafts.

References

[1] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *ACM/IEEE International Conference on SuperComputing SC 2002, Baltimore, USA*, November 2002.

IEEE COMPUTER SOCIETY

[2] Sun Microsystems Inc. RPC: Remote Procedure Call Protocol specification version 2. In *Tech. Rept. DARPA-Internet RFC 1057, SUN Microsystems, Inc.*, June 1988.

[3] Web Services. www.webservices.org.

[4] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1 (W3C Note). http://www.w3c.org/TR/2000/NOTE-SOAP-20000508/, 2000.

[5] Li Gong. JXTA: A Network Programming Environment. In *IEEE Internet Computing, Vol. 5, No. 3*, pages 88–95, http://java.sun.com/people/gong/papers/jxta-ieeeic.pdf, May/June 2001.

[6] I. Clarke, O. Sandberg, B. Wiley, , and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Springer, editor, *ICSI Workshop on Design Issues in Anonymity and Unobservability. Berkeley, California*, volume LNCS 2009, pages 46–66, July 2000.

[7] Andy Oram and Tim O'Reilly. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter 8 and 2. Edited by Andy Oram, March 2001.

[8] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. GridRPC: A Remote Procedure Call API for Grid Computing. In *Technical report, Univ. of Tennesse*, ICL-UT-02-06, June 2002.

[9] Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi. OmniRPC: A Grid RPC Facility for Cluster and Global Comuting in OpenMP. In Springer, editor, *Proc. of Workshop on OpenMP Applications and Tools 2001*, volume LNCS 2104, pages 130–135, West Lafayette, IN, USA, July 2001.

[10] Henri Casanova and Jack Dongarra. NetSolve: A Network-Enabled Server for Solving Computational Science Problems. In Sage Publications, editor, *The International Journal of Supercomputer Applications and High Performance Computing*, volume 11, Number 3, pages 212–223, 1997.

[11] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A Network Based Information Library for Globla World-Wide Computing Infrastructure. In Springer, editor, *Proc. of High-Performance Computing ans Networking, International Conference and Exhibition, HPCN Europe*, volume LNCS 1225, pages 491–502, Vienna, Austria, April 1997.

[12] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastrucure Toolkit. In MIT Press, editor, *The International Journal of Supercomputer Applications and High Performance Computing*, pages 115–128, Vol. 11, No. 2, 1997.

[13] Jean-Pierre Goux, Sanjeev Kulkarni, Jeff Linderoth, and Michael Yoder. An Enabling Framework for Master-Worker Applications on the Computational Grid. In IEEE Conputer Society, editor, *Proc. 9th IEEE Symp. on High Performance Distributed Computing*, pages 43–50, Pittsburgh, Pennsylvania, USA, 2000.

[14] Craig Lee and Domenico Talia. Grid Programming Models: Current Tools, Issues and Directions. In *Technical report, Indiana university, USA*, http://aspen.ucs.indiana.edu/CCPEwebresource/c618gridlee/c618Grid2002_LeeTalia.pdf, 2002.

[15] Java. http://java.sun.com.

[16] Meyer Bertrand. .Net Is Coming. In IEEE Computer Society, editor. volume 34, No. 8, pages 92–97, August 2001.

[17] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: A Generic Global Computing System. In IEEE Computer Society, editor, *IEEE Int. Symp. on Cluster Computing and the Grid*, pages 582–587, Brisbane, Australia. 2001.

IEEE COMPUTER SOCIETY